RÉSOLUTION DE L'ÉQUATION CUBIQUE PAR LA MÉTHODE DE STROBACH (PYTHON, C++)

PETER PHILIPPE

L'article « Solving cubics by polynomial fitting » écrit par Peter Strobach et publié en 2011 dans *Journal of Computational and Applied Mathematics*[1], présente un algorithme performant pour la résolution de l'équation cubique unitaire :

$$\mathcal{P}(x) = \prod_{i=1}^{3}(x - x_i) = (x - x_1)(x - x_2)(x - x_3) = x^3 + ax^2 + bx + c = 0$$

La méthode s'appuie sur une approche hybride permettant de décomposer cette équation via un produit de facteurs, un linéaire et un quadratique (dont le rôle sera crucial pour les racines complexes et les racines réelles multiples). Cela permet d'accroître considérablement la précision des racines obtenues et rend cette méthode plus efficace que les méthodes existantes, comme celle de Girolamo Cardano notamment et même par rapport à la méthode de Bairstow (bien que cette dernière ne soit pas cantonnée au degré 3). Elle montre également sa supériorité dans les cas où les racines sont très dispersées, ou inversement, lorsqu'elles sont très rapprochées. Ces quelques pages proposent un résumé assez simplifiée de la publication originale en ne reprenant que les grandes lignes, certains détails n'y figureront donc pas, il faudra alors se reporter à la publication originale téléchargeable librement à l'adresse citée au bas de cette page.

Tout d'abord, il est nécessaire de factoriser l'équation cubique telle que $\mathcal{P}(x) = x^3 + ax^2 + bx + c = (x^2 + \alpha x + \beta)(x + \gamma)$, avec $a = \alpha + \gamma$, $b = \beta + \alpha\gamma$ et $c = \beta\gamma$ où α, β et γ sont des inconnues.

Cette factorisation devient sous forme matricielle

$$\begin{pmatrix} 1 \\ a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 & & \\ \gamma & 1 & \\ & \gamma & 1 \\ & & \gamma \end{pmatrix} \begin{pmatrix} 1 \\ \alpha \\ \beta \end{pmatrix}$$

Puis, un vecteur d'ajustement $\mathbf{e}$ d'ordre 3 est créé :

$$\mathbf{e} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} - \begin{pmatrix} \gamma & 1 & 0 \\ 0 & \gamma & 1 \\ 0 & 0 & \gamma \end{pmatrix} \begin{pmatrix} 1 \\ \alpha \\ \beta \end{pmatrix}$$

1. https ://www.sciencedirect.com/science/article/pii/S0377042710006758

La matrice jacobienne $\mathbf{F}$ de ce système est :

$$\mathbf{F} = \begin{pmatrix} \frac{\partial}{\partial\alpha}\mathbf{e} & \frac{\partial}{\partial\beta}\mathbf{e} & \frac{\partial}{\partial\gamma}\mathbf{e} \end{pmatrix}^{\mathrm{T}} = -\begin{pmatrix} 1 & 0 & 1 \\ \gamma & 1 & \alpha \\ 0 & \gamma & \beta \end{pmatrix}$$

Les trois inconnues sont représentées via un vecteur $\mathbf{p} = \begin{pmatrix} \alpha & \beta & \gamma \end{pmatrix}^{\mathrm{T}}$. Après un développement avec la série de Taylor afin d'estimer l'erreur, conjugué à diverses opérations, l'auteur arrive au système linéaire suivant :

$$-\mathbf{F}\delta = \mathbf{e}$$

avec $\delta = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{pmatrix}$ un vecteur dont le rôle sera de mettre à jour les inconnues α, β et γ. La factorisation LU de la matrice jacobienne permet d'introduire deux nouveaux paramètres :

$$-\mathbf{F} = \begin{pmatrix} 1 & 0 & 0 \\ \gamma & 1 & \alpha \\ 0 & \gamma & \beta \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \gamma & 1 & 0 \\ 0 & \gamma & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & u_1 \\ 0 & 0 & u_2 \end{pmatrix}$$

avec $u_1 = \alpha - \gamma$ et $u_2 = \beta - \gamma u_1$.

La résolution du premier système matriciel issu de la factorisation LU amène à :

$$\begin{pmatrix} 1 & 0 & 0 \\ \gamma & 1 & 0 \\ 0 & \gamma & 1 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}$$

d'où

$$\begin{aligned} q_1 &= e_1 \\ q_2 &= e_2 - \gamma q_1 \\ q_3 &= e_3 - \gamma q_2 \end{aligned}$$

Le second système linéaire est alors :

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & u_1 \\ 0 & 0 & u_2 \end{pmatrix} \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

Sa résolution amène à obtenir les coefficients du vecteur δ :

$$\begin{aligned} \delta_3 &= q_3/u_2 \\ \delta_2 &= q_2 - u_1\delta_3 \\ \delta_1 &= q_1 - \delta_3 \end{aligned}$$

Ceci permet de mettre-à-jour les réels α, β et γ des deux facteurs (linéaire et quadratique) comme cela a été mentionné plus haut :

$$\begin{aligned} \alpha &= \alpha + \delta_1 \\ \beta &= \beta + \delta_2 \\ \gamma &= \gamma + \delta_3 \end{aligned}$$

Complétant ainsi l'ossature de cet algorithme. Toute équation de degré impair et à coefficients réels possède au moins une racine réelle [2], celle-ci est estimée par $\gamma = -\hat{x}_3$, puis les initialisations suivantes sont effectuées $\alpha = a - \gamma$, $\beta = b - \gamma\alpha$ et $e_3 = c - \gamma\beta$ de manière à annuler e_1, e_2 et e_3. Les initialisations suivantes sont dérivées de la méthode de Cardano $Q = \frac{a^2-3b}{9}$ et $R = \frac{2a^3-9ab+27c}{54}$, si la condition $R^2 < Q^3$ est vérifiée on pose $\Theta = \arccos\left(\frac{R}{Q^{3/2}}\right)$ permettant ainsi d'obtenir une première approximation pour $\hat{x}_3 = -2\sqrt{Q}\cos\left(\frac{\Theta}{3}\right) - \frac{a}{3}$, maintenant si la condition n'est pas vérifiée, on pose $A = -\text{signe}(R)\left(|R| + \sqrt{R^2 - Q^3}\right)^{1/3}$ puis $B = Q/A$ (ou bien $B = 0$ si $A = 0$), dans ce cas $\hat{x}_3$ sera égal à $\hat{x}_3 = A + B - \frac{a}{3}$.

De $\mathcal{P}(x) = (x^2 + \alpha x + \beta)(x + \gamma)$ avec $\alpha = -(x_1 + x_2)$, $\beta = x_1x_2$ et $\gamma = -x_3$, l'auteur enchaîne avec une série d'opérations permettant de traiter le cas des racines multiples (donc ici double et triple) afin d'aboutir finalement au déterminant de la matrice $-\mathbf{F}$:

$$|-\mathbf{F}| = x_1^2 + x_3^2 - 2x_1x_3$$

Car pour une équation cubique, trois cas peuvent se présenter selon la valeur du discriminant (trois racines réelles simples, une racine réelle triple et enfin une racine réelle simple et deux racines complexes conjuguées).

Définissons ensuite les célèbres relations de Viète pour notre équation cubique unitaire :

$$\begin{aligned} a &= -x_1 - x_2 - x_3 \\ b &= x_1x_2 + x_1x_3 + x_2x_3 \\ c &= -x_1x_2x_3 \end{aligned}$$

Après différentes substitutions, l'auteur construit une équation quadratique en x_2

$$x_2^2 + \frac{2a}{3}x_2 + \frac{b}{3} = 0$$

dont les solutions pour x_2 sont

$$x_2^{(1)} = -\frac{1}{3}\left(a + \text{signe}(a)\sqrt{a^2 - 3b}\right) \quad \text{et} \quad x_2^{(2)} = \frac{b}{3x_2^{(1)}}$$

et pour x_1 :

$$x_1^{(1)} = -a - 2x_2^{(1)} \quad \text{et} \quad x_1^{(2)} = -a - 2x_2^{(2)}$$

Seulement voilà, le discriminant nécessite l'emploi d'une valeur comme $x_1^{(1)}\left(x_2^{(1)}\right)^2$ ce qui n'est pas souhaitable. Ceci est contourné par lélaboration de deux discriminants :

$$\Delta_1 = c + \frac{(6b - 2a^2x_2^{(1)}) - ab}{9} \quad \text{et}\Delta_1 = c + \frac{(6b - 2a^2x_2^{(2)}) - ab}{9}$$

2. Ceci peut se prouver avec, entre autres, le Théorème des Valeurs Intermédiaires (alias le *TVI*), mais plus simplement en imaginant le fait qu'une valeur positive élevée au cube reste positive et qu'une valeur négative également élevée au cube restera négative, la fonction en question devra alors intersecter l'axe des abscisses au moins une fois pour être évaluée.

La détection d'une racine triple induit $a = -3x_1$, $b = 3x_1^2$, $c = -x_1^3$, elle est alors déduite de a par $x_1 = -\frac{a}{3}$.

L'auteur propose dans son article une implémentation en langage Fortran de son algorithme. Les implémentations perfectibles données ici le sont pour les langages C++ et Python, elles restent fidèles au code original. Des exemples identiques sont données dans les pages qui suivent pour les deux langages :

```cpp
#include <array>
#include <cmath>
#include <complex>
#include <format>
#include <expected>
#include <print>
#include <string>
#include <cstdlib>

void fitcs(double a, double b, double c, std::array<std::complex<double>, 3>& x) {
    const auto ap = a * a;
    const auto qq = (ap - 3.0 * b) / 9.0;
    const auto rr = (a * (2.0 * ap - 9.0 * b) + 27.0 * c) / 54.0;
    const auto qq3 = qq * qq * qq;
    const auto rr2 = rr * rr;
    const double inv3 = 1.0 / 3.0;
    double gamma;

    if (rr2 < qq3) {
        const auto theta = std::acos(rr / std::pow(qq, 1.5));
        const auto cos1 = std::cos(theta * inv3);
        gamma = std::fma(2.0 * std::sqrt(qq), cos1, a * inv3);
    } else {
        const auto aa = -std::copysign(std::pow(std::abs(rr) \
                        + std::sqrt(rr2 - qq3), inv3), rr);
        const auto bb = (aa == 0.0) ? 0.0 : qq / aa;
        gamma = -aa - bb + a * inv3;
    }

    double ee = 0.0;
    double alpha = a - gamma;
    double beta = b - alpha * gamma;
    double e1 = 0.0, e2 = 0.0, e3 = c - gamma * beta;

    for (int k = 0; k < 16; ++k) {
        double eee = ee;
        double eeee = eee;
        double u1 = alpha - gamma;
        double u2 = beta - gamma * u1;
        double q1 = e1;
        double q2 = e2 - gamma * q1;
        double q3 = e3 - gamma * q2;
        double delta3 = (u2 == 0.0) ? 0.0 : q3 / u2;
        double delta2 = q2 - u1 * delta3;
        double delta1 = q1 - delta3;

        alpha += delta1;
        beta += delta2;
        gamma += delta3;
```

```
        e1 = a - gamma - alpha;
        e2 = b - alpha * gamma - beta;
        e3 = c - gamma * beta;

        ee = std::fma(e1, e1, std::fma(e2, e2, e3 * e3));
        if (ee == 0.0 || ee == eee || ee == eeee)
            break;
    }

    const auto cc1 = alpha * 0.5;
    double diskr = std::fma(cc1, cc1, -beta);

    if (diskr >= 0.0) {
        diskr = std::sqrt(diskr);
        x[0] = {-cc1 - diskr, 0.0};
        x[1] = {beta / x[0].real(), 0.0};
    } else {
        diskr = std::sqrt(-diskr);
        x[0] = {-cc1, diskr};
        x[1] = {-cc1, -diskr};
    }

    x[2] = {-gamma, 0.0};
}

std::string formatx(const std::complex<double>& c) {
    return (c.imag() == 0.0) ? std::format("{}", c.real())
         : std::format("{}{:+}i", c.real(), c.imag());
}

std::expected<std::tuple<double, double, double>, std::string>
arguments(int argc, char* argv[]) {
    if (argc == 4) {
        try {
            double a = std::stod(argv[1]);
            double b = std::stod(argv[2]);
            double c = std::stod(argv[3]);
            return std::tuple{a, b, c};
        } catch (...) {
            return std::unexpected("Erreur de saisie !");
        }
    }
    return std::unexpected("Usage : ./fitcs_strobach a b c");
}

int main(int argc, char* argv[]) {
    auto result = arguments(argc, argv);
    if (!result) {
        std::println("Erreur : {}", result.error());
        std::exit(EXIT_FAILURE);
    }

    auto [a, b, c] = result.value();
    std::array<std::complex<double>, 3> roots;

    try {
        fitcs(a, b, c, roots);
```

```
        if (roots[0].real() == roots[1].real() \
        && roots[0].real() == roots[2].real()) {
            std::println("La racine triple est :\nx = {}", formatx(roots[0]));
        } else {
            std::println("Les racines sont :");
            for (std::size_t i = 0; i < roots.size(); ++i)
                std::println("x{} = {}", i + 1, formatx(roots[i]));
        }
    } catch (const std::exception& e) {
        std::println("Une exception est survenue : {}", e.what());
        std::exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}

% g++-14 -march=native -std=gnu++23 -pedantic -Wall -O3 strobach.cpp -o strobach
% % ./fitcs_strobach2 -6 11 -6
Les racines sont :
x1 = 2
x2 = 3
x3 = 1
% ./fitcs_strobach2 0 -7 6
Les racines sont :
x1 = 1
x2 = 2
x3 = -3
% ./fitcs_strobach2 7 11 5
Les racines sont :
x1 = -1
x2 = -1
x3 = -5
% ./fitcs_strobach2 -5 8 -4
Les racines sont :
x1 = 2
x2 = 2
x3 = 1
% ./fitcs_strobach2 -3 3 -1
La racine triple est :
x = 1
% ./fitcs_strobach2 0 6 -20
Les racines sont :
x1 = -1+3i
x2 = -1-3i
x3 = 2
% ./fitcs_strobach2 1 1 -39
Les racines sont :
x1 = -2+3i
x2 = -2-3i
x3 = 3
% ./fitcs_strobach2 -6 34 -104
Les racines sont :
x1 = 1+5i
x2 = 1-5i
x3 = 4
% ./fitcs_strobach2 1 0 -2
Les racines sont :
x1 = -1+1i
x2 = -1-1i
```

```
x3 = 1
```

```
import sys
import cmath
import math

def fitcubic(a: float, b: float, c: float) -> list[complex]:
    ap = a * a
    qq = (ap - 3.0 * b) / 9.0
    rr = (a * (2.0 * ap - 9.0 * b) + 27.0 * c) / 54.0
    qq3 = qq * qq * qq
    rr2 = rr * rr
    inv3 = 1.0 / 3.0

    if rr2 < qq3:
        theta = cmath.acos(rr / (qq ** 1.5))
        cos1 = cmath.cos(theta * inv3)
        gamma = 2.0 * float(cmath.sqrt(qq).real) * float(cos1.real) + a * inv3
    else:
        sqrt_val = cmath.sqrt(rr2 - qq3)
        aa = -math.copysign((abs(rr) + float(sqrt_val.real)) ** inv3, rr)
        bb = 0.0 if aa == 0.0 else qq / aa
        gamma = -aa - bb + a * inv3

    ee = 0.0
    alpha = a - gamma
    beta = b - alpha * gamma
    e1 = 0.0
    e2 = 0.0
    e3 = c - gamma * beta

    for _ in range(16):
        eee = ee
        eeee = eee
        u1 = alpha - gamma
        u2 = beta - gamma * u1
        q1 = e1
        q2 = e2 - gamma * q1
        q3 = e3 - gamma * q2
        delta3 = 0.0 if u2 == 0.0 else q3 / u2
        delta2 = q2 - u1 * delta3
        delta1 = q1 - delta3

        alpha += delta1
        beta += delta2
        gamma += delta3

        e1 = a - gamma - alpha
        e2 = b - alpha * gamma - beta
        e3 = c - gamma * beta
        ee = e1 * e1 + e2 * e2 + e3 * e3

        if math.isclose(ee, eee) or math.isclose(ee, eeee):
            break

    cc1 = alpha * 0.5
    diskr = cc1 * cc1 - beta
```

```
    roots = [0j, 0j, 0j]
    if float(diskr.real) >= 0.0:
        sqrt_d = cmath.sqrt(diskr)
        x0 = complex(-cc1 - float(sqrt_d.real), 0.0)
        x1 = complex(beta / x0.real, 0.0)
        roots[0] = x0
        roots[1] = x1
    else:
        sqrt_d = cmath.sqrt(-diskr)
        roots[0] = complex(-cc1, float(sqrt_d.real))
        roots[1] = complex(-cc1, -float(sqrt_d.real))

    roots[2] = complex(-gamma, 0.0)
    return roots

def format_complex(c: complex) -> str:
    real = round(c.real, 6)
    imag = round(c.imag, 6)
    if imag == 0:
        return f"{real}"
    elif imag < 0:
        return f"{real}{imag}i"
    else:
        return f"{real}+{imag}i"

def main():
    if len(sys.argv) != 4:
        print("Usage : python3 script.py a b c")
        return

    try:
        a = float(sys.argv[1])
        b = float(sys.argv[2])
        c = float(sys.argv[3])
    except ValueError:
        print("Erreur : les arguments doivent être des nombres réels.")
        exit

    try:
        roots = fitcubic(a, b, c)
        if roots[0].real == roots[1].real == roots[2].real:
            print("La racine triple est :")
            print(f"x = {format_complex(roots[0])}")
        else:
            print("Les racines sont :")
            for i, root in enumerate(roots, 1):
                print(f"x{i} = {format_complex(root)}")
    except Exception as e:
        print(f"Une erreur est survenue : {e}")

if __name__ == "__main__":
    main()

% python3 strobach.py -6 11 -6
Les racines sont :
x1 = 2.0
x2 = 3.0
x3 = 1.0
```

```
% python3 strobach.py 0 -7 6
Les racines sont :
x1 = 1.0
x2 = 2.0
x3 = -3.0

% python3 strobach.py 7 11 5
Les racines sont :
x1 = -1.0
x2 = -1.0
x3 = -5.0

% python3 strobach.py -5 8 -4
Les racines sont :
x1 = 2.0
x2 = 2.0
x3 = 1.0

% python3 strobach.py -3 3 -1
La racine triple est :
x = 1.0

% python3 strobach.py 0 6 -20
Les racines sont :
x1 = -1.0+3.0i
x2 = -1.0-3.0i
x3 = 2.0

% python3 strobach.py 1 1 -39
Les racines sont :
x1 = -2.0+3.0i
x2 = -2.0-3.0i
x3 = 3.0

% python3 strobach.py -6 34 -104
Les racines sont :
x1 = 1.0+5.0i
x2 = 1.0-5.0i
x3 = 4.0

% python3 strobach.py 1 0 -2
Les racines sont :
x1 = -1.0+1.0i
x2 = -1.0-1.0i
x3 = 1.0
```